Representación y Manipulación de la Información en Conjuntos de Instrucciones

Miguel Angel Astor Romero

Versión 2 - 8 de septiembre de 2016

Introducción

Formalmente, una computadora digital es una máquina electrónica general diseñada para ejecutar instrucciones provistas por el usuario [1]. Estas se componen por múltiples elementos electrónicos, siendo los más importanes el CPU (*Central Processing Unit* - Unidad Central de Procesos), la memoria y los dispositivos de entrada/salida.

El componente fundamental de la computadora es el CPU , el cual se compone a su vez por varios módulos, siendo los más importantes la unidad de control y la ruta de datos, también conocida como unidad aritmético-lógica [1]. La ruta de datos realiza operaciones sobre números binarios, con los cuales es posible representar prácticamente cualquier dato o información [2] La memoria se utiliza para almacenar datos e instrucciones. Las instrucciones son cargadas por el usuario en la memoria junto a los datos utilizados por estas. Un conjunto de instrucciones y datos forman una estructura llamada programa, y las computadoras que funcionan de esta manera son conocidas como Computadoras de Programa Almacenado, o Computadoras de Arquitectura de Von Neumann [3].

El resto de este documento está estructurado de la siguiente manera. En la Sección 1 se describen los fundamentos para el almacenamiento y manipulación de datos arbitrarios en la memoria de una computadora. La Sección 2 presenta distintas formas de representación concretas para varios tipos de datos básicos elementales. La Sección 3 muestra como se realizan las operaciones aritméticas básicas en binario sobre números enteros. En la Sección 4 se describe como se representan las instrucciones en una computadora digital. Finalmente se presentan las conclusiones.

1. Representación de la información

En el nivel electrónico más bajo las computadoras modernas operan sobre señales eléctricas mediante dispositivos microscópicos llamados transistores [2]. Estas señales toman voltajes dentro de un rango que se encuentra entre los -0,5 y los 5,5 voltios. Las señales se dividen en dos sub-rangos de voltaje llamados bajo y alto como puede observarse en la

Figura 1. Las señales de salida se emiten dentro de rangos de voltajes más reducidos que las señales de entrada para reducir el impacto del ruido en dichas señales [2]. Estos dos rangos permiten codificar información discreta como dígitos binarios, donde algunas señales corresponden al uno y las señales opuestas corresponden al cero, o viceversa. La elección de que rangos corresponden al cero y al uno es arbitraria.

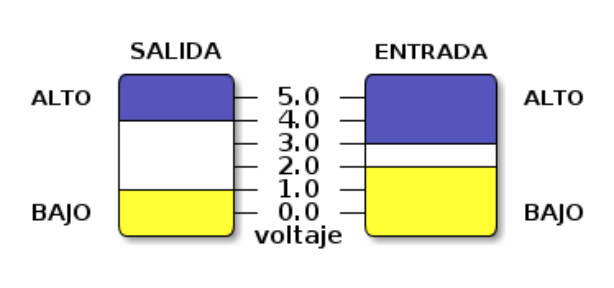

Figura 1: Rangos de voltaje para señales binarias. Figura recuperada de [2].

1.1. Bases numéricas

Los sistemas de numeración representan los números como una serie de sumas de productos de dígitos multiplicados por potencias de la base del sistema de numeración. En el sistema decimal la base es el número diez y en el sistema binario la base es el dos. Un ejemplo de esto se puede observar en la Ecuación 1, donde se puede ver la descomposición del número 724,5 en términos de potencias de 10.

$$724{,}5 = 7 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 + 5 \times 10^{-1} \tag{1}$$

De forma general un sistema de numeración de base b utilizará b dígitos diferentes y un número en dicho sistema se representará como una sumatoria de productos de dichos dígitos por potencias de b [2].

1.1.1. Números binarios

El sistema de numeración binario utiliza el dos como base. Según lo establecido anteriormente, esto quiere decir que los números en este sistema se componen únicamente por dos dígitos los cuales se construyen mediante la suma de productos de dichos dígitos con potencias de base 2. Por convención se utilizan los mismos símbolos arábigos del sistema decimal para representar el uno y el cero para representar los dígitos disponibles al sistema binario [2]. Un solo dígito binario es conocido como un bit, y una agrupación ordenada de bits (usualmente ocho bits pero esto puede variar [1]) es conocida como un *byte* . Un conjunto ordenado de ocho bits es también conocido como un octeto [4]. Utilizando conjuntos arbitrariamente grandes de bits es posible representar cualquier conjunto de datos discreto [2]

La Ecuación 2 muestra la representación del número $1010{,}11_2$, y su valor en decimal. Como se puede observar, para realizar la conversión de un número binario a decimal basta con sumar las potencias de dos donde el bit correspondiente tiene el valor uno. En este documento un subíndice b indica la base del número en cuestión. Se asume que un número sin subíndice está en base decimal.

$$1010{,}11_2 = 1\times2^3+0\times2^2+1\times2^1+0\times2^0+1\times2^{-1}+1\times2^{-2} = 8+2+0{,}5+0{,}25 = 10{,}75 \quad (2)$$

Para realizar la operación inversa, es decir, convertir un número de decimal a binario se aplica un algoritmo sencillo [2]. Primero se identifica la potencia de dos más grande tal que restada al número decimal de un resultado positivo. El procedimiento se aplica de manera recursiva, restando la potencia de dos más grande al resultado de la resta anterior hasta que la diferencia sea 0. Con esto se obtienen las componentes de base dos del número decimal. El número binario correspondiente se obtiene tomando los coeficientes de la serie de potencias respectiva a la suma de las componentes obtenidas. Un ejemplo de esto se puede observar en la Ecuación 3, donde se aplica el algoritmo mencionado al número 49.

$$\begin{aligned} R_1 &= 49-32 = 17 & 32 &= 2^5 \\ R_2 &= R_1-16 = 1 & 16 &= 2^4 \\ R_3 &= R_2-1 = 0 & 1 &= 2^0 \\ 625 &= 2^9+2^6+2^5+2^4+2^0 = 1001110001_2 \end{aligned} \quad (3)$$

1.1.2. Números octales y hexadecimales

El sistema de numeración octal utiliza el ocho como base. Se suelen utilizar los dígitos arábigos del cero al siete para representar números en este sistema. Por su parte, el sistema hexadecimal utiliza la base 16. Como solo existen 10 dígitos arábigos, se utilizan por convención las primeras seis letras del alfabeto romano para representar los dígitos faltantes [2]. De esta forma las letras A, B, C, D, E, y F corresponden a los valores 11, 12, 13, 14 y 15 respectivamente. Estos sistemas de numeración se utilizan porque permiten representar números binarios de tres y cuatro bits respectivamente de una manera compacta.

1.2. Precisión

La memoria de una computadora no es infinita y debe poseer una cierta estructura para poder utilizarse. En general, las memorias de las computadoras digitales modernas se componen de un arreglo lineal de celdas las cuales son agrupaciones de una cierta cantidad de bits, usualmente de un *byte* de longitud como estándar *de facto* [1]. Cada celda de la memoria está asociada a otro número binario llamado dirección el cual se utiliza para indexar cada celda en particular.

La cantidad de bits de memoria que la computadora lee o escribe con una única operación se conoce como el tamaño de palabra de la computadora y usualmente es un múltiplo de

ocho [1]. Las implicaciones de esto se detallan en la Sección 1.3. Este tamaño determina la cantidad de bits sobre los que operan las instrucciones de la computadora. Por ejemplo, una computadora con palabras de 32 bits puede leer o escribir cuatro *bytes* por palabra y sus instrucciones operan sobre datos de 32 bits. Cabe resaltar que en las computadoras basadas en la arquitectura IA32 (*Intel Architecture* - Arquitectura Intel) y, por extensión, AMD64 (*Advanced Micro Devices* - Micro Dispositivos Avanzados), por razones históricas se llama palabra a un conjunto de 16 bits [5].

Como se puede intuir, el tamaño de la palabra determina la cantidad de elementos que pueden representarse con un número binario compuesto por dicha cantidad de bits. Para cada distinto tipo de dato esta cantidad, modificada por la técnica de representación o codificación del mismo (véase la Sección 2) se conoce como la precisión del mismo [6]. De forma general, dado un tamaño de palabra de b bits se podrán representar a lo sumo 2^b elementos de un conjunto discreto arbitrario. Es posible tener datos de precisión mayor al tamaño de la palabra, en cuyo caso se combinan palabras sucesivas.

1.3. Direccionamiento de bytes

Por diversas razones resulta impráctico el direccionar la memoria bit a bit [3]. En lugar de esto las computadoras digitales suelen direccionar la memoria *byte* a *byte* o palabra a palabra. Como se mencionó anteriormente, cada celda de memoria se asocia a una dirección. Al ser las memorias arreglos lineales de celdas sus direcciones corresponden entonces a una serie creciente de números enteros que van desde el cero hasta un valor máximo. Al igual que cualquier otro número dentro de la computadora, las direcciones de memoria se representan como números binarios los cuales tienen una longitud en bits específica. Esta longitud determina la cantidad de *bytes* direccionables en la memoria. Es importante notar que el tamaño de la palabra no determina el tamaño de las direcciones de la memoria, siendo perfectamente posible tener, por ejemplo, computadoras con palabras de 64 bits que indexan la memoria con direcciones de 48 bits [7].

Una dirección de memoria que coincide con el inicio de una palabra se dice que es una dirección alineada [3]. Debido a que las computadores se diseñan para leer o escribir datos a memoria en bloques de una palabra, entonces se hace necesario ubicar los datos en la memoria de manera que puedan ser obtenidos o modificados con una sola instrucción de lectura/escritura. Esto se conoce como alineación de los datos [3].

Algunas computadoras como las basadas en los procesadores IA32 permiten realizar lecturas o escrituras a memoria en direcciones de *byte* arbitrarias, mientras que otras como la arquitectura 68000 de Motorola solo permiten realizar lecturas/escrituras en direcciones alineadas [3].

1.4. Ordenamiento de bytes

Los *bytes* dentro de una palabra pueden ser ordenados de derecha a izquierda o viceversa. Cuando el ordenamiento se realiza de izquierda a derecha se dice que la computadora en

cuestión es de tipo *big endian*, en caso contrario se dice que es *little endian* [1]. Estos distintos ordenamientos de *bytes* pueden observarse en la Figura 2.

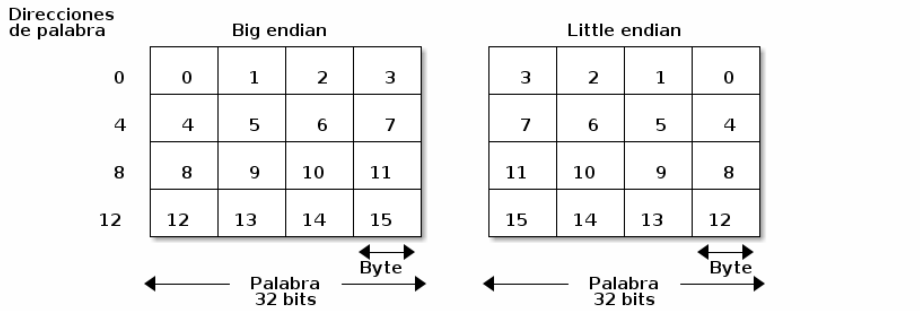

Figura 2: Formas de ordenamiento de *bytes*. Figura recuperada de [1].

El uso de un ordenamiento u otro tiene sus efectos sobre la representación de los datos que tienen una precisión de múltiples *bytes* [1]. Tomemos por ejemplo la representación del número decimal $2870136116 = AB12CD34_{16}$, asumiendo que el número está almacenado a partir de la dirección de byte 100. Las representaciones *big endian* y *little endian* correspondientes pueden observarse en la Figura 3. El principal efecto del ordenamiento de *bytes* se observa al transferir datos como este entre ambos tipos de computadoras [1]. Transferir los *bytes* uno a uno de la computadora *big endian* a la *little endian* provocará que los *bytes* sean recibidos en el orden inverso al esperado, provocando que el número sea interpretado como $34CD12AB_{16} = 885854891$. Para evitar este problema es necesario invertir los *bytes* de las palabras ya sea antes de enviarlas o al recibirlas siempre y cuando el tipo de dato que se está transmitiendo posea una precisión mayor a un byte.

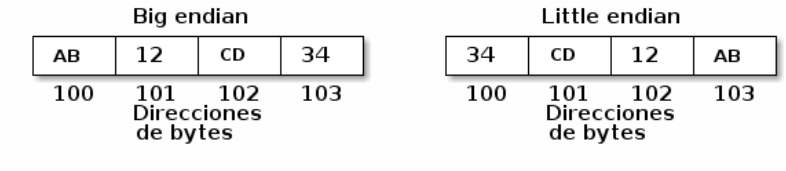

Figura 3: Representación del número $AB12CD34_{16}$ con distintos ordenamientos de *bytes*.

2. Representación de datos

Siendo los números binarios dentro de la computadora agrupaciones finitas de bits, se pueden utilizar estos bits de distintas formas para asociar números binarios específicos con elementos de otros conjuntos finitos [2]. Este procedimiento de asociación se conoce como representación o codificación de los datos [6]. En esta Sección se detallan distintas maneras de

codificar datos numéricos y de texto, los cuales son datos discretos. Igualmente se presentan dos formas de asociar el conjunto de los números reales con los números binarios.

2.1. Enteros

Los números enteros pueden representarse de dos formas, sin signo o con signo. Los enteros sin signo se representan en de forma directa, interpretando los números binarios sin codificación [6]. Para el caso de los números negativos se pueden utilizar diferentes codificaciones.

2.1.1. Signo-Magnitud

Esta representación utiliza el primer bit del número binario para representar el signo del mismo, interpretándose los demás bits como la magnitud. La representación signo-magnitud se define como se observa en la Ecuación 4 [6], siendo x un número binario de n bits y x_{n-1} el bit más a la izquierda de x (es decir, el bit más significativo). Esta codificación tiene la propiedad de que el cero tiene dos representaciones como ± 0.

$$R_n(\overrightarrow{x}) = (-1)^{x_{n-1}} \times \left(\sum_{i=0}^{n-2} x_i 2^i \right) \tag{4}$$

2.1.2. Complementos a 1 y a 2

El complemento es una operación realizada por las computadoras para simplificar las operaciones de resta y manipulación lógica [8]. Para representar enteros utilizando complementos se definen dos tipos de representaciones conocidas como el complemento a la base y el complemento a la base disminuido. De forma general se define el complemento a la base b para un número N de n dígitos en base b como se observa en la Ecuación 5 [8].

$$C_b(N) = \begin{cases} b^n - N & N \neq 0 \\ 0 & N = 0 \end{cases} \tag{5}$$

Por su parte el complemento a la base disminuido se define como se observa en la Ecuación 6 [8].

$$C_{b-1}(N) = (b^n - 1) - N \tag{6}$$

Sustituyendo las bases en las definiciones anteriores podemos ver que para el caso de los números binarios el complemento a la base y el complemento a la base disminuido definen el complemento a 2 y el complemento a 1 respectivamente. El complemento a 1 de un número binario de n bits se puede obtener invirtiendo el valor de cada bit [8], mientras que el complemento a 2 se obtiene calculando el complemento a 1 y luego sumando 1 al valor resultante [6].

En estas representaciones se interpreta el primer bit del número como el signo del mismo, siendo el cero el indicador de signo positivo y el uno el indicador de signo negativo. De esta forma, se pueden observar las siguientes propiedades [6]:

1. En complemento a 1 el cero tiene dos representaciones como ± 0.
2. En complemento a 2 solo hay una representación para el cero.
3. En complemento a 2, el complemento del número negativo de menor valor es el mismo número negativo.

2.1.3. BCD

BCD (*Binary Coded Decimal* - Decimal Codificado a Binario) es un sistema de codificación que permite asociar cada uno de los 10 dígitos del sistema decimal con un número binario [8]. Esta codificación se realiza asociando un patrón de cuatro bits a cada dígito como se observa en la Tabla 1. Como se puede observar cada dígito decimal se codifica utilizando el número binario correspondiente.

Tabla 1: Codificación BCD. Tabla recuperada de [8].

Dígito decimal	**Dígito BCD**
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001

Un ejemplo de un número decimal codificado con BCD puede verse en la Ecuación 7 junto a su representación en binario. Todo número decimal codificado con BCD requerirá de $4 \times k$ bits, siendo k la cantidad de dígitos del número original en decimal.

$$185_{10} = 000110000101_{bcd} = 10111001_2 \tag{7}$$

2.2. Caracteres y cadenas de caracteres

La representación de textos dentro de una computadora se lleva a cabo mediante mecanismos de codificación de caracteres, en los cuales cada carácter se asocia de forma única a un número binario. Existen múltiples mecanismos de codificación de caracteres, siendo los más

utilizados el código ASCII (*American Standard Code for Information Interchange* - Código Americano Estándar para el Intercambio de Información) y el estándar Unicode [5].

2.2.1. ASCII y ASCII extendido

El código ASCII es una codificación de caracteres alfanuméricos utilizada para representar 128 caracteres como números binarios de 7 bits. De estos 128 caracteres, 94 representan caracteres imprimibles mientras que 34 representan caracteres de control [2]. Al utilizar solo 7 bits por carácter se hace particularmente útil para la representación de textos sin las dificultades introducidas por los distintos esquemas de ordenamiento de bytes descritos en la Sección 1.4; sin embargo, este código no funciona para la representación de textos en otros idiomas aparte del inglés.

Como usualmente un *byte* corresponde a ocho bits, esto implica que un carácter codificado con ASCII poseerá un bit sobrante al ser almacenado en un *byte*. Por lo general este bit suele ser utilizado con el valor cero; Sin embargo, es posible utilizar este bit para extender la codificación de caracteres [2]. Usualmente se conoce como ASCII extendido a cualquier codificación basada en el estándar ASCII que utilice el bit ocho para este proposito. Sin embargo, no existe una extensión estándar al código ASCII, siendo cualquier codificación identificada como ASCII extendido una tecnología propietaria [5].

2.2.2. Unicode

Unicode es un estándar de codificación de texto ampliamente utilizado para representar textos en una enorme gama de lenguajes [5]. Fue diseñado por el consorcio Unicode y utiliza un conjunto de caracteres de 32 bits conocido como UCS (*Universal Coded Character Set* - Conjunto de Caracteres Codificados Universal) el cual contiene representaciones de más de un millón de caracteres [6]. UCS no se utiliza directamente, sin embargo, siendo más común en los sistemas computacionales moderno el uso de las codificaciones alternativas UTF-8 y UTF-16. Estas representaciones son variantes concretas del mecanismo de codificación de Unicode conocido como UTF (*Unicode Transformation Format* - Formato de Transformación de Unicode) [9].

1. UTF-8

   La codificación UTF-8 fue creada en 1992 por Rob Pike y Ken Thompson para el sistema operativo distribuido Plan 9 [9]. Fue diseñada para garantizar compatibilidad directa con sistemas que utilizan la codificación ASCII. El ocho en el nombre de esta codificación se deriva del hecho de que la unidad mínima de valor (la cantidad de bits mínima posible utilizada para representar un carácter) es de ocho bits. La versión más reciente de UTF-8 es capaz de representar todos los caracteres representables por UTF-16 por razones de compatibilidad [10].

   Esta codificación se caracteriza por utilizar de uno a cuatro octetos por carácter con las siguientes propiedades [10]:

a) Todo carácter de un único octeto corresponde exactamente con el respectivo carácter ASCII para el mismo octeto.

b) Los caracteres de más de un octeto corresponden a caracteres representables con UTF-16.

c) Cada carácter tiene una codificación única.

La Tabla 2 resume el proceso de codificación de caracteres en UTF-8. La columna de la izquierda contiene los rangos de caracteres representados según con cada cantidad de octetos, y la columna de la derecha representa la distribución de los bits dentro de cada octeto. Para realizar la codificación primero se busca en la primera columna el rango al que pertenece el carácter según su numero asignado en UCS. Luego se rellenan los bits marcados con x en la Tabla con los bits de la representación en binario del número del carácter, colocando el bit de orden menor del número de carácter en el bit de orden menor del último octeto.

Tabla 2: Codificación de octetos en UTF-8. Tabla recuperada de [10].

Rango de caracteres (en hexadecimal)	**Secuencia de octetos UTF-8** (en binario)
0000 0000-0000 007F	0xxxxxxx
0000 0080-0000 07FF	110xxxxx 10xxxxxx
0000 0800-0000 FFFF	1110xxxx 10xxxxxx 10xxxxxx
0001 0000-0010 FFFF	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Como la unidad de valor mínima de UTF-8 es de un solo octeto, no es necesario tomar precauciones para evitar problemas de ordenamiento de *bytes* [10].

2. UTF-16

Esta codificación utiliza dos ó cuatro octetos para representar los caracteres de UCS [11]. La codificación tiene las siguientes características:

a) Un carácter con valor menor a 10000_{16} se representan en un entero de 16 bits cuyo valor es igual al número correspondiente al carácter en UCS.

b) Los caracteres con valores entre 10000_{16} y $10FFFF_{16}$ se representan por un entero de 16 bits cuyo valor está entre $D800_{16}$ y $DBFF_{16}$, seguido de otro entero de 16 bits cuyo valor está entre $DC00_{16}$ y $DFFF_{16}$.

c) Los caracteres con valor mayor a $10FFFF_{16}$ no son representables.

d) Los valores entre $D800_{16}$ y $DFFF$ están reservados y no se utilizan para representar caracteres.

UTF-16 define al carácter $FEFF_{16}$ como un carácter especial conocido como la BOM (*Byte Order Mark* - Marca de Ordenamiento de *Bytes*). La BOM se utiliza cuando un texto codificado con UTF-16 debe ser enviado por una red, de forma que el receptor pueda identificar el ordenamiento de *bytes* utilizado por el transmisor [11].

2.3. Reales

A pesar de que el conjunto de los números reales no es un conjunto discreto, es posible codificarlo de forma muy limitada en números binarios [6]. Existen dos formas de lograr esta codificación: la representación de coma fija y la de coma flotante.

2.3.1. Reales de coma fija

La representación de coma fija utiliza un concepto del sistema de numeración binario conocido como la coma binaria [2], la cual es análoga a la coma decimal en el sistema de numeración decimal y se utiliza para separar la parte entera de la parte fraccionaria de un número. La ecuación 2 muestra un número real en binario con coma binaria y su equivalente decimal. Dado un dato binario de precisión finita, se establece una cierta cantidad de bits para representar la parte entera del número a representar, y otra porción para representar la parte fraccionaria [3].

Como se puede intuir, este mecanismo de representación se caracteriza por poseer una precisión muy limitada. Sin embargo, se suele utilizar dado que la ejecución de operaciones aritméticas es mucho más eficiente con esta representación que con las representaciones de coma flotante, debido a que se pueden utilizar las mismas operaciones de aritmética entera para complemento a 2 (véase la Sección 3) con los números reales de coma fija [3]. De hecho, la representación de enteros con complemento a 2 es un caso especial de la representación de reales con coma fija, asumiendo que la coma binaria se encuentra a la derecha del último bit del número [3].

2.3.2. Reales de coma flotante

En 1985 el IEEE (*Institute of Electrical and Electronic Engineers* - Instituto de Ingenieros Eléctricos y Electrónicos) publicó el estándar IEEE 754 el cual define una codificación de punto flotante comúnmente conocida como *IEEE floating point* [6]. Cabe resaltar que tanto esta como otras representaciones de coma flotante solo pueden representar correctamente números racionales que puedan ser escritos de la forma $x \times 2^y$. Cualquier otro valor solo puede ser aproximado [6].

De forma concreta, el estándar IEEE 754 representa números de la forma $V = (-1)^s \times M \times 2^E$ donde:

- s determina el signo del número. Se representa con un solo bit.
- M es el significando o mantisa.
- E es el exponente.

Estos elementos se representan en binario empaquetados dentro de una palabra de 32 bits (conocida como precisión simple) o 64 bits (precisión doble) como se ve en la Figura 4. En la Figura el valor e corresponde a la codificación del exponente y f a la codificación de la mantisa como se verá a continuación.

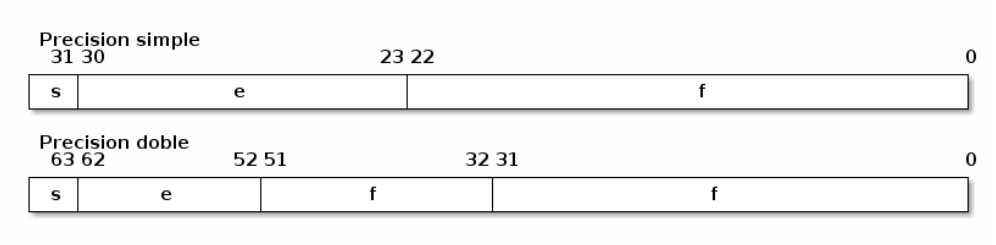

Figura 4: Formatos coma flotante IEEE 754. Figura recuperada de [6].

Para codificar un número utilizando el estándar IEEE 754 se siguen reglas diferentes dependiendo del valor del campo del exponente [6].

1. Caso 1: valores normalizados

   Si e es distinto de cero, o distinto de 255 en precisión simple o 2047 en precisión doble se considera este caso. En este caso se interpreta al exponente como un entero con signo en forma sesgada, donde el sesgo se calcula como $2^{k-1} - 1$ siendo k el número de bits del campo del exponente. El valor final del exponente se calcula como $E = e - sesgo$.

   La mantisa se interpreta como un número fraccional de la forma $f = 0.f_{n-1}f_{n-2}\ldots f_0$, donde n es la cantidad de bits del campo de la mantisa y el bit f_{n-1} corresponde con el bit más a la izquierda del campo. El valor final de la mantisa será $M = 1 + f$.

   Los valores normalizados se utilizan para representar números reales cuyo valor absoluto sea mayor o igual a uno.

2. Caso 2: valores denormalizados.

   Si el campo del exponente tiene todos los bits en cero se considera este caso. Los valores denormalizados se utilizan para representar números cuyo valor absoluto sea menor a uno incluyendo el cero, el cual tiene dos representaciones dependiendo del bit de signo.

   En este caso, el exponente se calcula como $E = 1 - sesgo$ con el sesgo a su vez calculado de la misma forma que con los valores normalizados. Por su parte la mantisa se calcula como $M = f$.

3. Caso 3: valores especiales

   Si el campo del exponente tiene todos los bits en uno se considera este caso. Los valores especiales se utilizan únicamente para codificar tres valores:

   a) El infinito positivo se representa con el bit de signo en cero y todo el campo de la mantisa en cero.

   b) El infinito negativo se representa igual que el infinito positivo pero con el bit de signo en uno.

c) Si el campo de mantisa tiene uno o más bits en uno independientemente del bit de signo se representa el valor NaN (*Not a Number* - No es un Número), el cual se usa para identificar codificaciones no válidas.

3. Aritmética binaria de números enteros

Las operaciones aritméticas elementales (suma, resta, multiplicación y división) siguen las mismas reglas cuando se utiliza el sistema binario que en el sistema decimal [8]. Sin embargo, como se describió en la Sección 2.1, los números enteros se tienen que representar de alguna forma de manera que la computadora sea capaz de distinguir números con signo y sin signo, aunado al hecho de que la computadora utiliza una cantidad de bits finita para representar dichos números. Esto trae con sigo dos problemas a resolver: como realizar dichas operaciones tomando en cuenta la codificación de los números y que sucede cuando la precisión de la computadora no es suficiente para almacenar el resultado de las operaciones.

3.1. Aritmética en complemento a 2

La aritmética en complemento a 2 sigue las mismas reglas de la aritmética binaria [8]. Al sumar números en complemento a 2 se debe sumar también el bit de signo, descartando cualquier acarreo que se produzca en este. La Ecuación 8 muestra un ejemplo de suma en complemento a 2. El valor en decimal de cada cantidad se da entre paréntesis.

$$\begin{array}{r} (6)00000110_2 \\ +(-13)11110011_2 \\ \hline (-7)11111001_2 \end{array} \tag{8}$$

Para realizar la resta de dos números en complemento a 2, primero se toma el complemento a 2 del sustraendo y luego se suma este al minuendo [8]. Un ejemplo de esto se puede observar en la Ecuación 9.

$$\begin{array}{r} (6)00000110_2 \\ -(13)00001101_2 \\ \hline (-7)11111001_2 \end{array} \equiv \begin{array}{r} (6)00000110_2 \\ +(-13)11110011_2 \\ \hline (-7)11111001_2 \end{array} \tag{9}$$

La multiplicación en complemento a 2 es equivalente a la multiplicación entera normal [6], y puede ser utilizada también para multiplicar números sin signo. Un ejemplo de multiplicación binaria se puede ver en la Ecuación 10.

$$\begin{array}{r} (11)1011_2 \\ \times \quad (5)0101_2 \\ \hline 1011_2 \\ 0000_2 \\ 1011_2 \\ \underline{0000_2} \\ (55)00110111_2 \end{array} \tag{10}$$

Para el caso de la división entera, la división entre potencias de dos es equivalente a realizar un desplazamiento lógico de un bit a la derecha [6]. El desplazamiento lógico a la derecha corresponde a mover todos los bits del número desplazado una cierta cantidad de posiciones hacia la derecha, repitiendo el bit de signo por la izquierda. De la misma forma, desplazar los bits de un número una posición hacia la izquierda es equivalente a multiplicar por una potencia de dos [6].

3.2. Overflow

Cuando la precisión no es lo suficientemente grande para almacenar un resultado ocurre lo que se conoce como un desbordamiento u *overflow* [6]. Es decir, si se tienen dos números x, y tales que $0 \leq x, y \leq 2^w - 1$, entonces estos dos números pueden ser representados utilizando w bits. Al sumar ambos números el resultado puede estar en el rango $0 \leq x + y \leq 2^{w+1} - 2$, el cual puede necesitar hasta $w + 1$ para almacenarse. Un ejemplo de esto puede verse en la Ecuación 11 donde se observa la suma de dos números sin signo de cuatro bits cada uno.

$$\begin{array}{r} (13)1101_2 \\ +(11)1011_2 \\ \hline (24)11000_2 \end{array} \tag{11}$$

Para el caso de la multiplicación sin signo, el producto de dos números x, y bajo las mismas condiciones mencionadas caerá en el rango $0 \leq x \cdot y \leq 2^{2w} - 2^{w+1} + 1$ el cual puede necesitar hasta $2w$ bits para almacenarse [6]. Siendo el tamaño de dato utilizado para almacenar el resultado de w bits entonces los bits sobrantes del resultado son descartados, tanto en el caso de la suma como de la multiplicación. Esta operación se conoce como truncamiento [6]. Debido a esto, la suma de la Ecuación 11 tiene como resultado $1000_2 = 8$.

En el caso de la suma en complemento a 2 se pueden presentar dos casos particulares conocidos como desbordamiento positivo y desbordamiento negativo [6]. El desbordamiento positivo ocurre cuando se suman dos números positivos y se obtiene un resultado negativo, mientras que el desbordamiento negativo ocurre cuando se suman dos números negativos y se obtiene un resultado positivo.

4. Representación del conjunto de instrucciones

Como se mencionó en las Secciones anteriores, las instrucciones que puede ejecutar la computadora son tratadas igualmente como datos y almacenadas en memoria según el concepto de Programa Almacenado de la Arquitectura de Von Neumann. Las instrucciones en si y el como se representan difiere considerablemente entre distintos fabricantes de procesadores. Sin embargo, de forma general una instrucción de computadora debe contener por lo menos un código de operación y hasta una, dos o tres referencias a memoria [3].

Las instrucciones de una sola referencia a memoria hacen uso implícito de un registro de procesador (una unidad de memoria que contiene una sola palabra y se ubica físicamente en el procesador) conocido como el acumulador. La siguiente es un ejemplo de este tipo de instrucciones, la cual suma el valor contenido en la dirección de memoria A al valor almacenado en el acumulador y guarda el resultado en el acumulador.

$$sumar A$$

Las instrucciones de dos referencias a memoria pueden utilizar de forma explícita dos registros de procesador o utilizar simultáneamente una dirección de memoria y un registro. Esta posibilidad de poder utilizar distintos tipos de referencias a memoria en una instrucción se conoce como modo de direccionamiento [3]. Uno de estos dos operandos actúa a su vez de acumulador o destino de la instrucción [3]. Un ejemplo de estas instrucciones es la siguiente, la cual mueve el contenido de la dirección de memoria A al registro R_1, siendo este el destino de la operación.

$$cargar A, R_1$$

Finalmente, las instrucciones de tres referencias a memoria utilizan explícitamente dos operandos y un destino para cada instrucción [3]. Un ejemplo de esto es la instrucción siguiente, la cual suma el contenido de los registros R_1 y R_2, almacenando el resultado en el registro R_3. Independientemente de la cantidad de referencias a memoria que utilicen las instrucciones, las arquitecturas de procesadores modernas usualmente solo permiten que las operaciones aritmético-lógicas sean realizadas con registros de procesador como operandos [3].

$$sumar R_1, R_2, R_3$$

El conjunto de todas las instrucciones que puede ejecutar la computadora se conoce como su ISA (*Instruction Set Architecture* - Arquitectura de Conjunto de Instrucciones). Comercialmente se distinguen dos formas de diseñar la ISA de una computadora, en base a la cantidad de instrucciones que esta posea. Estas se conocen como las arquitecturas RISC (*Reduced Instruction Set Computer* - Computadora de Conjunto de Instrucciones Reducido) y CISC (*Complex Instruction Set Computer* - Computadora de Conjunto de Instrucciones Complejo) [1].

4.1. RISC

Los procesadores basados en la arquitectura RISC se caracterizan por tener un conjunto pequeño de instrucciones, usualmente al rededor de 50 aunque esto puede variar [1]. Estas instrucciones se seleccionan e implementan de forma que puedan ser iniciadas muy rápidamente. De igual forma, al tener menos instrucciones se simplifica el diseño del procesador, lo que se traduce en un menor número de transistores (el componente electrónico fundamental de un procesador). Debido a esto, los procesadores RISC son notables por consumir menos energía y producir menos calor que los procesadores CISC, razón por la cual son muy utilizados en el entorno de sistemas embebidos [1]. Ejemplos de procesadores RISC son las arquitecturas ARM (*Advanced RISC Machine* - Máquina RISC Avanzada) de ARM Holdings y SPARC (*Scalable Processor Architecture* - Arquitectura de Procesadores Escalable) de Oracle [3].

4.2. CISC

En contra parte, los procesadores CISC poseen conjuntos de instrucciones extensos, usualmente con más de 250 o 300 instrucciones. La ventaja de estos procesadores con respecto a los procesadores RISC es que pueden representar operaciones más complejas con menos instrucciones [1]. Sin embargo, esto se traduce en procesadores más complejos que consumen más energía que los procesadores RISC [1]. De igual forma, debido a la circuitería más compleja de los procesadores CISC las instrucciones en estos son más lentas de iniciar. Hoy en día los procesadores CISC modernos utilizan un núcleo RISC el cual se encarga de ejecutar las instrucciones de uso más común del procesador de forma que se pueda obtener el aumento de desempeño que implica el menor tiempo de inicio de instrucción de los procesadores RISC, dejando las instrucciones menos comunes a un núcleo CISC [1]. Ejemplos de procesadores CISC son las arquitecturas IA32 de Intel, AMD64 de AMD y 68000 de Motorola [3].

Conclusiones

En este documento se realizó un estudio detallado sobre como las computadoras digitales representan la información. Se definió a una computadora digital como una máquina electrónica capaz de realizar operaciones sobre números binarios. Estos números binarios se almacenan en una memoria, la cual es un arreglo lineal ordenado de números binarios de ocho dígitos conocidos como *bytes*.

Utilizando números binarios es posible codificar cualquier información arbitraria. Se estudiaron en detalle esquemas para la representación de números enteros con y sin signo, números reales y cadenas de caracteres. Así mismo, se presentó un repaso acerca de la ejecución de operaciones aritméticas sobre números enteros utilizando el complemento a 2 (una de las técnicas de representación de enteros con signo estudiadas) junto al problema del desbordamiento que puede presentarse al tratar de realizar aritmética con números binarios de precisión finita.

Finalmente se mostró como se representan las instrucciones que ejecutan las computadoras digitales, instrucciones que son tratadas como datos según el modelo de Programa Almacenado de la Arquitectura de computadoras de Von Neumann, prevalente en el diseño de computadoras digitales modernas. Se mostraron las distintas formas de codificar instrucciones con uno, dos o tres operandos, así como dos formas estándar de estructurar estos conjuntos de instrucciones conocidas como las arquitecturas RISC y CISC.

Referencias

[1] A. Tanenbaum, *Organización de Computadoras: Un Enfoque Estructurado*, 4^a Edición, Prentice Hall, 2000.

[2] M. Mano y C. Kime, *Fundamentos de Diseño Lógico y de Computadoras*, 3^a Edición, Pearson, 2005.

[3] C. Hamacher, Z. Vranesic y S. Zaky, *Organización de Computadores*, 5^a Edición, McGraw Hill, 2003.

[4] D. Comer, *Redes Globales de Información con Internet y TCP/IP Vol. 1: Principios Básicos, Protocolos y Arquitectura*, 3^a Edición, Prentice Hall, 1996.

[5] K. Irvine, *Lenguaje Ensamblador para Computadoras Basadas en Intel*, 5^a Edición, Prentice Hall, 2008.

[6] R. Bryant y D. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 2^a Edición, Pearson, 2011.

[7] Anónimo, *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, Revisión 3.22, Advanced Micro Devices, 2012.

[8] M. Mano, *Diseño Digital*, 3^a Edición, Pearson, 2003.

[9] R. Pike y K. Thompson, *Hello World or*, Proceedings of the Winter 1993 USENIX Conference, USENIX Association, pp. 43-50, Berkeley, 1993.

[10] F. Yergeau, *UTF-8, a transformation format of ISO 10646*, IESG Internet Engineering Steering Group, RFC 3629, 2003.

[11] P. Hoffman y F. Yergeau, *UTF-16, an encoding of ISO 10646*, IESG Internet Engineering Steering Group, RFC 2781, 2000.